Playing Super Hexagon using Computer Vision

Dipl.-Ing. Christoph Wiesmeier BSc

Institute of Computer Graphics and Vision
Graz University of Technology, Austria

Seminar Paper
Advisor: Dipl.-Ing. Dominik Narnhofer BSc
Supervisor: Univ.-Prof. Dipl.-Ing. Dr.techn. Thomas Pock
Graz, April 1, 2021

Contact: Christoph Wiesmeier, c.wiesmeier@student.tugraz.at

Abstract

The introduction and advances in CNNs have enabled reinforcement learning to be a viable option for computer vision. In this work we use a computer game as a substitute for a real world-system. The game is played from pixel values and runs without time synchronization. We compare a simple traditional computer vision implementation, with a supervised learning system, on our reinforcement learning system. Additionally we study the influence of image preprocessing to our learning-based systems. Finally we show how our SARSA based reinforcement implementation achieved superhuman performance in the game Super Hexagon

Keywords: *Technical report, ICG, Reinforcement Learning, SARSA, CNN*

1 Introduction

Computers are capable of solving more and more tasks. But are they able to play completely unmodified computer games, and how hard is it to implement such a system? To answer this question we decided to implement three different algorithms to play the game SuperHexagon. A traditional computer vision, a supervised learning and a reinforcement learning approach. All of them play an unmodified computer game which we consider a substitute for a real-world system. As we don't synchronize the execution speed of the game to our algorithm, the resulting system is comparable to a camera capturing a real world system and the computer moving some leaver. This is in contrast to most current reinforcement systems like ([2][5]) where the environment does not have any state transitions while the algorithm is planing its next steps.

In Section 2 we will discuss the similarities and differences to some related works. This then leads us to Section 3 where we will explain the game choice. In Section 4 we will discuss our software framework and associated tools. At the end of this section we then introduce our preprocessing pipeline of which we will use parts for all our implementations. Section 5 introduces our Traditional Computer Vision (TCV) implementation which provides us with baseline results. After discussing the benefits and limitations of this approach we will move towards learning based solutions. A supervised system is introduced in Section 6. We evaluate the supervised system to get a good starting point for our final approach. In Section 7 we introduce our reinforcement learning system. We discuss our implementation of State Action Reward State Action (SARSA) and evaluate the results. Finally we will compare the results of the different approaches in Section 8

2 Related Work

In recent years a lot of work was done in the field of reinforcement learning. The most common introduction in this field is from Barto and Sutton [12]. It provides the de facto standard for basics, notation and problem formalization in reinforcement learning.

With the introduction of Convolutional Neural Networks (CNNs) as a function approximation in reinforcement learning, it becomes possible to run the algorithms on pixel images. A very popular task in this area is playing pong from pixels with a lot of implementations like [8],[9],[10]. Pong is one of many Atari games which are used regularly to develop and compare vision-based reinforcement systems. These Atari games are provided as part of OpenAIs Gym [2] toolkit. Challenges like ViZDoom [5] extend this to the area of 3D games. Hafner [3] shows some successes in ViZDoom but struggles to provide the amount of training required for high quality results.

All approaches, mentioned so far, synchronize the game's execution speed to the reinforcement agent. Ramstedt and Pal [11] analyze the differences of these synchronized systems with real-time systems. A system called Reactive Reinforcement Learning is proposed by [13]. This approach reorders some tasks to reduce latency, where our implementation parallelizes them.

We tried at some point to preload the replay buffer with existing data to improve the initial performance. We stopped our investigations as the initial results were not promising and the task was outside of our focus. More sophisticated approaches to pre-training with promising results are discussed in Hester et al.[4].

Furthermore there are optimizations and other algorithms which we did not implement but they could be interesting for similar tasks. The implementation of a Dueling Q-Network [14] could improve the stability of your system. There is a trend towards Policy Gradient and Actor Critic methods [12],[7] which could be an alternative to replace our SARSA algorithm.

3 The Game

For this work we let the computer play Super Hexagon [1]. In this game the player, a small triangle, can rotate around the center of the screen. From the outside hexagons are shrinking, moving towards the center. These hexagons have some open sides. The player has to move around the center to avoid a collision with the hexagon's closed sides. The view on the game-field is rotating as well as mild perspective transformations distort the view. An example screenshot can be seen in Figure 1. Using the left and right arrow keys the player tries to survive as long as possible. The main challenge for a human player is the game's speed.

We have chosen this game because of its simplicity. The objective is clearly defined as maximizing the survival time. The game rounds are quite short. A human player usually achieves records of around 5 to 8 seconds during the first games. After a few hours of playing, records of about 60 to 70 seconds are typical. Furthermore the simple graphic allows to analyze the frames in real-time with a reasonable amount of processing power.

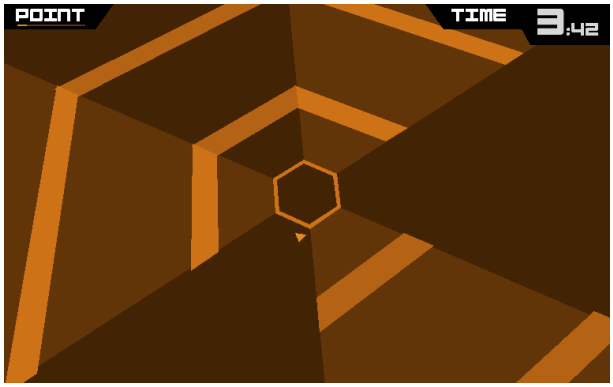


Figure 1: Example image of the game Super Hexagon.

4 Framework

To allow the computer to play a game, more than just the control algorithm is needed. In this section we will explain the basics of additionally required framework.

SuperHexagon, like most computer games, is designed to be played by humans. Because we want to control the game by a program, we have to create an Application Programming Interface (API) around it (Wrap the Game). Further we need to navigate through the menus of the game and detect when we have lost. To store and inspect the game rounds we also need a recording format and a viewer. Finally we implement an image preprocessing which simplifies the work of the control algorithms.

4.1 Wrapping the Game

To allow interaction with the game we built a wrapper class. Its main purpose is to start and stop the game and give access to the game's inputs and outputs. Because the game is started as a normal window application with no control over its execution speed, implementing a standard interface like GYM[2] is not possible.

The Linux version of Super Hexagon is based on the Steam platform. By creating a "steam_appid.txt" file the game can be started directly. This simplifies getting the process ID and also allows starting more than one instance at the same time.

[1] https://superhexagon.com/

Figure 2: Template example used to detect failed games. The background is marked as transparent and ignored during matching.

The video output of the game is captured using screen grabbing. Therefore we locate the window using its Process ID and grab the image using the library MSS [2]. This approach only works for visible windows. There are applications like Firefox which also are capable of grabbing background applications, but we could not find a Python library with this option. Against our initial approach of using the game in Full-Screen mode we switched over to window mode. This reduces the image resolution and as a consequence the required processing power.

Initially the input emulation was done by using PyUserInput [3] but later we switched to PyXdo [4]. This limited our application to Linux systems with X-Server but allows sending the inputs to a specific process rather than the current window with focus. This is a major improvement as now we can run multiple game instances at once and also other applications can still be used on the same PC.

We also shortly investigated the option of using a virtual machine to run the game. From performance perspective it looks possible, but decided against it to avoid unnecessary complexity.

4.2 Game Logic

We need to implement a basic game logic to be able to start a game and understand when the game-episode is lost. Since we only have the visual output of the game we extract the information from the images. This is done by a simple template matching. The templates use transparency to mask out image regions. An example template can be seen in Figure 2. The matching between the image s and the template t is defined in Equation 1-3. There are templates for StartScren, Death, Failed Menu and Difficulty Selection. The template matching happens directly after the screen grabbing. The matched template is added as an annotation to the frame when sent for processing. For performance reasons the template matching is only done at 1/4 of the resolution.

$$s \in \mathbb{N}^{768\times480\times3}\,,\; t \in \mathbb{N}^{192\times120\times4}\,,\; x \in \mathbb{N}\,|\,x < 192\,,\; y \in \mathbb{N}\,|\,y < 120 \tag{1}$$

$$diff(x,y) = \begin{cases} \max\limits_{c=\{0,1,2\}} (abs(s[x*4, y*4, c] - t[x,y,c]), & \text{if } t[x,y,3] \neq 0 \\ 0, \quad \text{otherwise} \end{cases} \tag{2}$$

$$\text{match} : \frac{1}{\underset{\forall(x,y)}{count}(t[x,y,3] \neq 0)} \sum_{\forall(x,y)} diff(x,y) < 55 \tag{3}$$

[2] https://github.com/BoboTiG/python-mss
[3] https://github.com/SavinaRoja/PyUserInput/wiki/Installation
[4] https://github.com/rshk/python-libxdo

The game logic automatically starts the game as required by the current task. The implemented game logic is developed with robustness in mind and needs to be able to recover from different states. Sometimes the game does not react immediately to inputs. This is most probably a result of timing issues.

While a game is running, the frames are forwarded to a player object, which analyzes the images and selects a move. This player object can be a Human Player or an algorithm like a reinforcement agent.

4.3 Recording

All grabbed game frames and their annotations like the matched template or the decision of the player are recorded. There are two types of recordings created. The debug-recording containing all frames, including menus, is used for development. The per-game-round-recording which only contains the frames of one round is used for evaluations and to building data-sets for learning systems.

The recordings are HDF5 [5] files. Each frame is stored as a HDF5 data-set object. These objects contain the PNG compressed images as data and the collected annotations as attributes. The PNG compression is done on the fly in a thread pool to avoid performance loss in the main execution path while still keeping memory consumption low. The recording takes on average 23kB/frame. This typically results in around 10MB per game, largely deepening on the playtime.

4.4 Recording Analyzer

To understand which decisions our algorithms have taken and why, we need a way to inspect the recordings and check the behavior of new algorithms. This we achieve with our recording analyzer (Figure 3). It loads a recording and allows seeking to an arbitrary position. So we can inspect the attributes, export images, or even debug the TCV algorithm on the current frame using Pythons debugger.

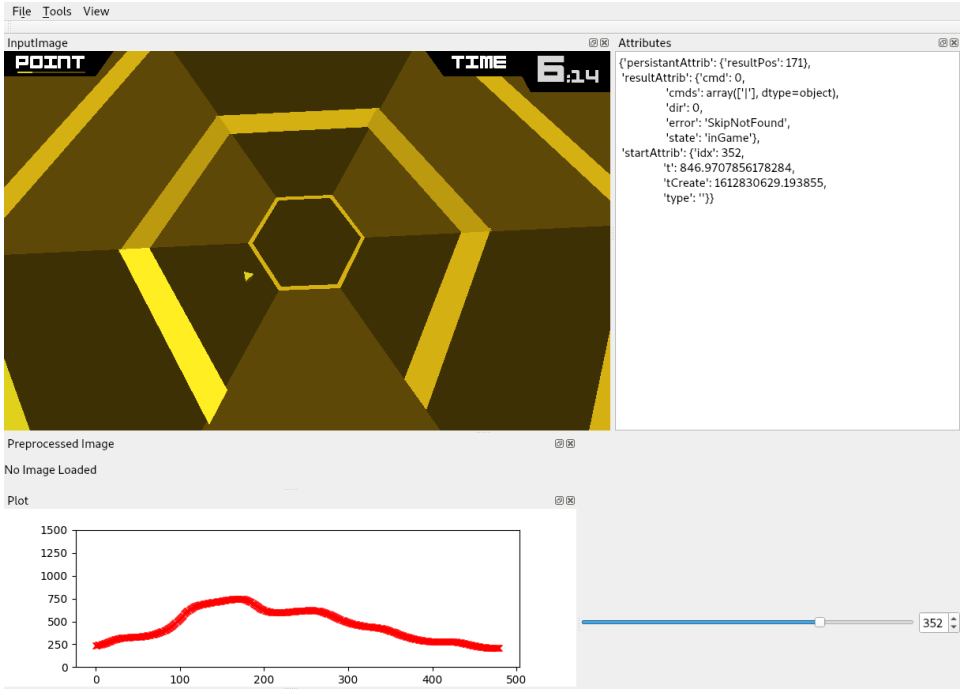


Figure 3: Screenshot of the recording analyzer.

[5] https://www.hdfgroup.org/solutions/hdf5

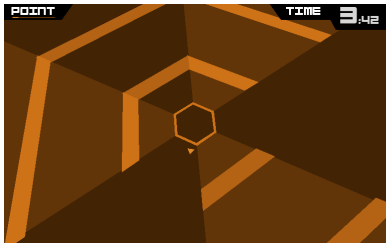
4.5 Preprocessing

The image contains more information than necessary to play the game. During preprocessing we remove unnecessary information, like color, and try to optimize the representation for our control algorithms. The TCV algorithm requires all preprocessing steps. The learning-based implementations also uses parts of the processing pipeline.

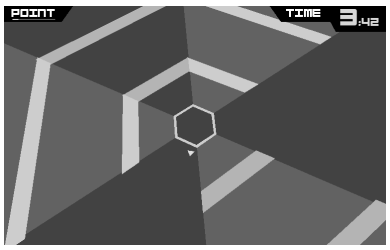
The first preprocessing step is gray-scaling. Color information is not necessary to play the game. Reducing the image to only one channel simplifies further processing. The conversion is done by transforming into Hue Saturation Value (HSV) and then only using the Value channel. The result is visible in Figure 4b. This preprocessing step is referred to as *gray* in the later algorithms.

The second step is transforming the image to polar coordinates. This gives us position and a distance as coordinates, which are simpler to handle. As can be seen in Figure 4c this also removes the menus but it loses a significant portion of the image. This preprocessing step is referred as *rad* in later algorithms.

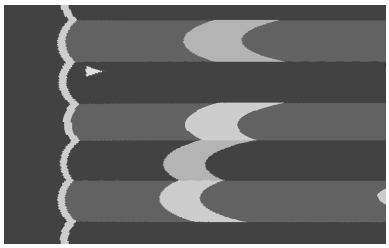
The next step is centering the player. This is done by first stacking two *rad* images (Figure 4d). Next the player is searched using OpenCV's[1] blob detector (Detections are shown as red circles in Figure 4e). The detections are filtered based on their distance on the X-Axis. From remaining detections the point nearest to the center of the Y-Axis is selected. Finally the image is cropped with the selected blob in the vertical center (Figure 4f). This preprocessing step is referred as *radStab* in later algorithms.



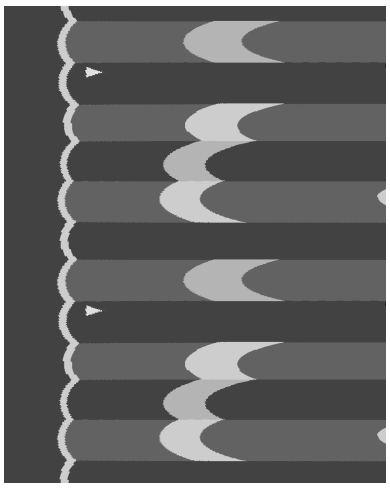
(a) Input image



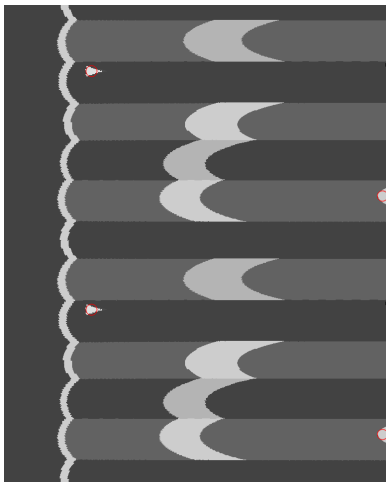
(b) Grayscaled imput image, refered as *gray*



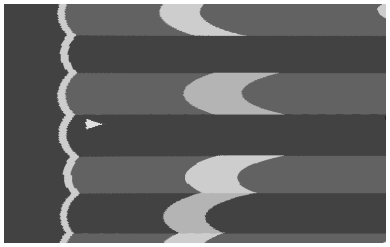
(c) Image in polar cordinates, refered as *rad*



(d) Stacked ploar image



(e) Image with detected blobs



(f) Final cropped image refered as *radStab*

Figure 4: Overview of different preprocessing steps.

5 Traditional Computer Vision

The first implementation of an artificial agent for playing the game is a non-learning based one. It is heavily based on the preprocessing pipeline and adds some additional steps.

The *radStab* image (Figure 4f) is thresholded by 60% from its maximum value. A collision with objects to the left of the player is not longer possible. Therefore the image is cut on the player's vertical position with a small additional margin ($xCut$). If the margin is big enough the player is not longer on the image and can't be detected as an obstacle. Finally the most right column is set to a non-zero value to make sure each line has an object. The result is shown in Figure 5a.

We define the index i_a for the angular dimension and the index i_d for the distance dimension. By applying the argmax (index of the maximum) the obstacle distance $d_{init}(i_a)$ is calculated for each possible player position (Equation 6, Figure 5b top left). A weighting function $w(i_a)$ is defined acording to Equation 7 (Figure 5b bottom left). The weighting is subtracted form the distances resulting in the weighted distance $d_w(i_a)$ (Equation 9). This $d_w(i_a)$ gives priority to the nearest option, which has a high distance (Figure 5b top right). The corners of the objects have always the highest distance of this object. This would lead to target solutions directly on edge of a collision. To give priority to the center of the gap, the distances d_w are filtered with a gaussian distribution $h(x)$. The final dinstance signal $d(i_a)$ (Equation: 10) is show in (Figure 5b bottom right).

The target position is now the position with the largest free distance. Depending on the target position the command (CMD) Left, Right or Straight/Neutral direction is selected. For Straight there is a small dead-zone around the center where, the non-optimal position is accepted. This avoids unnecessary moves, which sometimes lead to accidental collisions on low frame rates.

$$i_a \in \mathbb{N} \,|\, i_a \leq I_a, I_a = 480\,, \quad i_d \in \mathbb{N} \,|\, i_d \leq I_d \,|\, I_d \in \mathbb{Z}, I_d \leq 768 \tag{4}$$

$$img \in \mathbb{N}^{I_d \times I_a} \,|\, img \leq 255 \tag{5}$$

$$d_{init}(i_a) = \underset{i_d}{argmax}(img) \tag{6}$$

$$w(i_a) = -\left(abs(i_a - I_a/2) - I_a/2\right) * ca \,|\, ca \in \mathbb{R}^+ \tag{7}$$

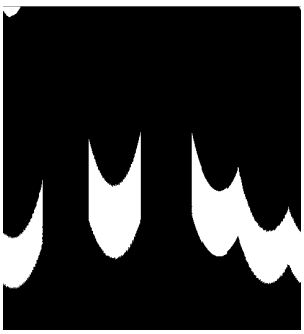
$$h(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{x^2}{2\sigma^2}} \,|\, x \in \mathbb{Z} \tag{8}$$

$$d_w(i_a) = d_{init} - w \tag{9}$$

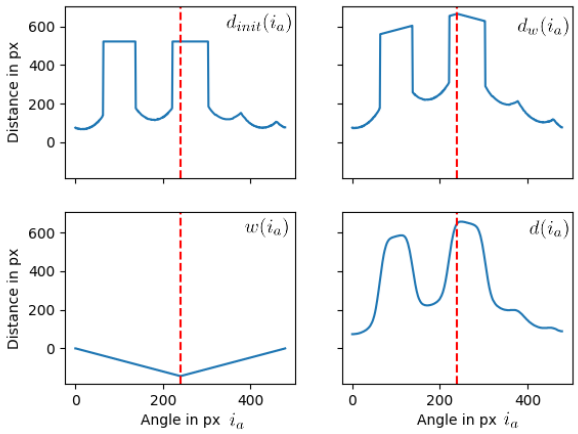
$$d(i_a) = d_w * h \tag{10}$$

$$CMD = \begin{cases} \text{"}L\text{"}, \underset{i_a}{argmax}(d) < I_a/2 - 25 \\ \text{"}R\text{"}, \underset{i_a}{argmax}(d) > I_a/2 + 25 \\ \text{"}N\text{"}, \text{otherwise} \end{cases} \tag{11}$$

The system has a lot of parameters which need to be adjusted. For three of them we performed a parameter search. *XCut* is the number of pixels removed to the right of the player after thresholding. Center adjust(ca) is a gain factor for our distance weighting. Gaussian (σ) is the standard deviation of the gaussian filter on the final distance function. In Figure 6 we run each configuration for 100 rounds and then compare the results. For Center adjust(ca) and Gaussian (σ) we see a sweet spot of the parameters. For $xCut$ there is no clear trend, we only see 200px is too much. Based on our tests we finally choose *gaussian*=10, *center adjust*=0.6, xCut=70. We tired to avoid further tests in this area as they are very time intensive. Creating Figure 6 requires playing 2200 game rounds, with about 20 seconds each, and therefore it takes approximately 12 hours.



(a) Cropped and thresholded image. Rotated 90° for Illustration.



(b) TL: Callculated distances, TR: combined, BL: Distance weigthing, BR: Gaussian filtered

Figure 5: Processing steps of the TCV algorithm.

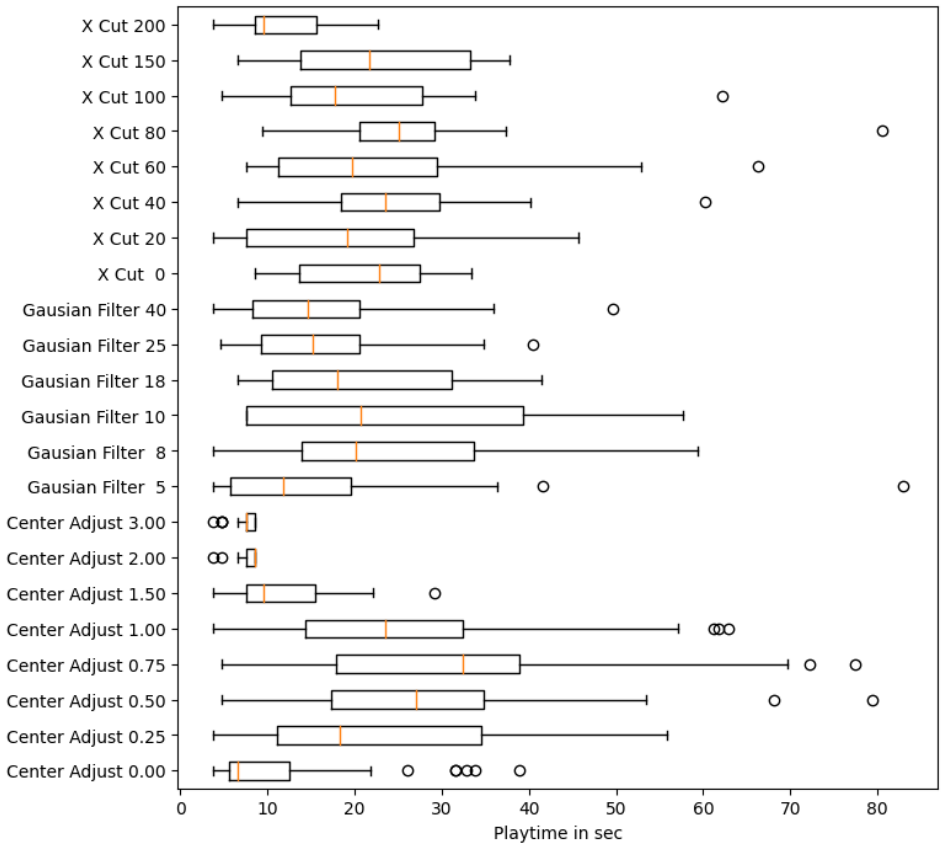


Figure 6: Variation of parameters of the TCV algorithms. Each configuration was tested for 100 rounds.

5.1 Results

For evaluation of our TCV implementation we compare it with a random policy and a human. The random policy is the ε-greedy policy from the reinforcement algorithm with $\varepsilon = 1$. We run each implementation for 100 rounds and compare their playtime. The result can be seen in Figure 7 and Table 1. The average performance of our TCV algorithm is slightly better than the human reference, but the difference is not very significant.

	Min	Mean	Median	Max
Random	0.6	2.1	1.4	13.9
Human	2.1	23.1	22.6	64.1
TCV	2.4	24.4	23.5	60.2

Table 1: Results of 100 rounds of different algorithms.

Beyond the systematic limitations of the algorithm, which we will discuss in the next section, we noticed a significant influence of the frame-rate on the performance. For execution speeds below 25 frames per second (fps) we noticed a significant reduction in achieved performance. A large factor for this reduction is an unavoidable lag when using an external system. The state of the system changes during the processing time. The selected action is sent to the game approximately at the same time as the next frame is captured. Additionally the game also has a reaction time. To illustrate this, if action a_n is a result of processing frame n, we observe the corresponding change only in frame $n + 2$ or $n + 3$.

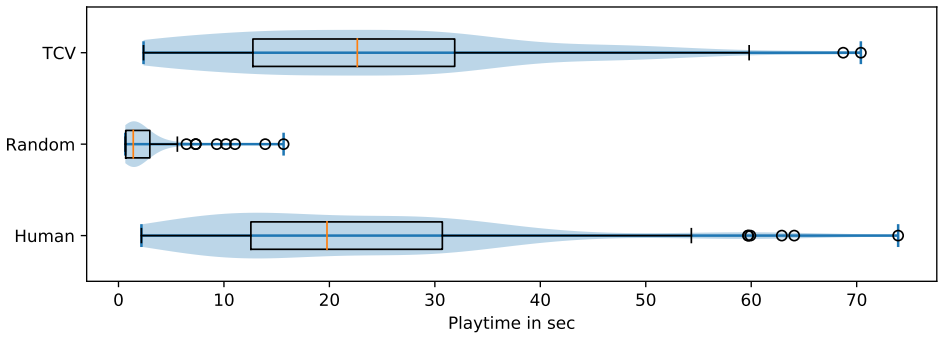


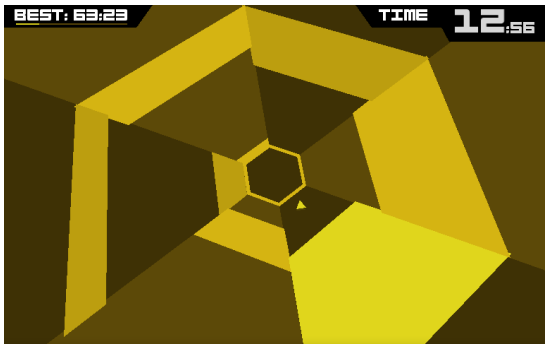
Figure 7: Overview of the distribution of playtimes over 100 rounds per algorithm as a violin plot stacket on a boxplot.

5.2 Common Errors

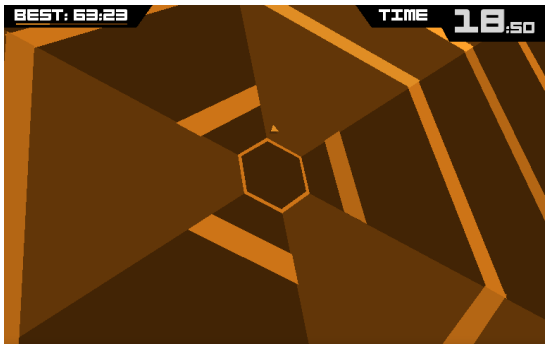
The implementation makes some systematic mistakes. It always tries to get to the best position on the shortest path. It does not perform any checks if this path is actually possible or may lead to a collision. Figure 8 shows examples where taking the direct path resulted in a collision.

To overcome the issue in example 1a, a path-finding algorithm would work. This is computationally difficult and could significantly reduce the frame-rate of the system. To also solve example 2 the movement speed needs to be considered. Unfortunately the speed of the approaching objects changes during the game which makes this difficult. A simple speed estimation and collision avoidance was tested during the development. Unfortunately it never worked reliably.

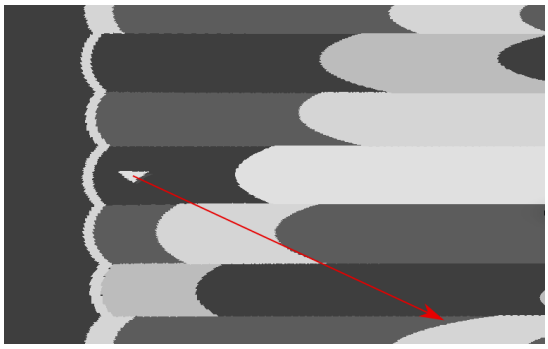
The approach of tracking down possible movement paths or even a low-resolution path-finding would still be quite promising. A significant challenge could be achieving a good performance especially while implementing it in Python.



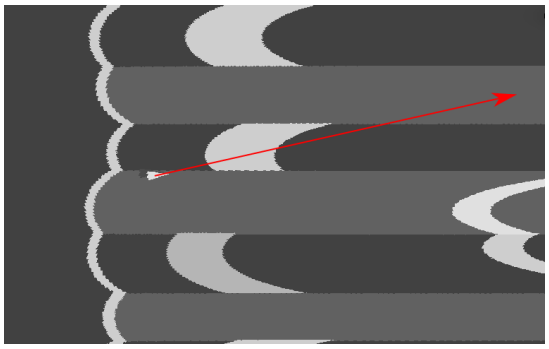
(a) Input Image (Example 1)



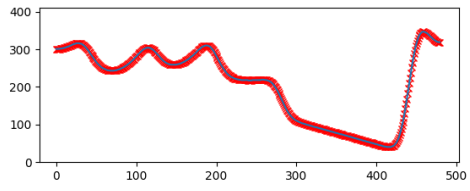
(b) Input Image (Example 2)



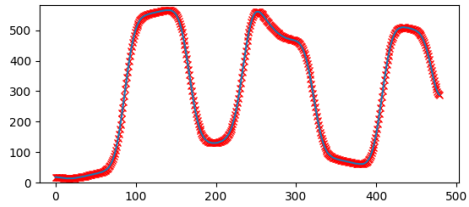
(c) Sabilized Image (Example 1)



(d) Sabilized Image (Example 2)



(e) Distances (Example 1)



(f) Distances (Example 2)

Figure 8: In Example 1 the algorithm detected the optimal position to be on the lower end of the image. The shortest move would be down. The objects in between are not considered by the algorithm.
In Example 2 an upward position would have a slightly higher distance. The player is moved upwards but can't reach its target position before it hits an object.

6 Supervised Learning

The TCV implementation needs a lot of tuning. Also handling each systematic error is a lot of effort. Therefore we next evaluate the option of supervised learning. For us supervised learning is mostly as a precursor for reinforcement learning, but it also can be an option to learn from expert players, which by far exceed the typical human performance.

Our supervised learning system is implemented using Tensorflow/Keras. It is trained using data from a human player, the TCV implementation, or a combination of both. The data-sets are shown in Table 2.

We treat the task of playing the game as an image classification problem. The input is a prepocessed screen image, the output is one of three possible actions. The input preprocessing can be gray scaling Figure 4b (gray), polar transformation Figure 4c (rad), or image stabilization Figure 4f (radStab). A CNN is used as a function approximation ($f(x_i)$) and convertes the input image (x_i) into one hot encoded output (y_i).

$$f(x_i) -> y_i \quad |0 \leq x_i \in \mathbb{R}^{192\times120\times1} \leq 1, \quad y_i \in \mathbb{R}^3 \tag{12}$$

Initially we considered using a single output to map the options to $left = -1$, $straight = 0$, $right = 1$. This was dropped during initial testing, as there are many situations where it does not matter, if left or right is chosen, but the neutral command would lead to a collision.

For all CNNs a dropout regularization is used for the fully connected layers. A probability of 0.2 is used for all hidden layers and 0.05 for the output layer. The convolutional layers are not regularized, as they are less likely to overfit. For training we minimize the Cross Entropy Loss (CEL) using Adam [6].

Equation 13 defines the $CEL(i)$ for each sample i as baed on the one hot encoded target label y_i^* and the CNN output $f(x_i)$. We typically use $Mean(CEL)$ which shows the average overall N samples of an epoch. Figure 9 shows the $Mean(CEL)$ and mean classification error ($mean(E)$) during training. The evaluation on the dev-/test-set is performed on each fifth epoch. The loss indicates slight overfitting, but the classification error shows no significant issue. We use 50 epochs to allow a convergence for different network architectures without tuning the hyperparameters for each configuration.

$$CEL(i) = -\sum_{c=0}^{3} y_i^* log(f(x_i)) \quad | f(x_i), y_i \in \mathbb{R}^{C=3} \tag{13}$$

$$mean(CEL) = \frac{1}{N}\sum_{i=0}^{N} CEL(i) \tag{14}$$

$$mean(E) = \frac{1}{N}\sum_{i=0}^{N} \begin{cases} 1\,, \text{ if } \underset{c}{argmax}(t_i) \neq \underset{c}{argmax}(y_i^*) \\ 0, \text{otherwise} \end{cases} \tag{15}$$

6.1 Evaluation

We train the algorithm on different data-sets shown in Table 2. The *Human A* data-set was created at the beginning of our work. It contains a few long game sessions and it is not split into rounds. The *Human100* data-set was initially created as a performance reference for all algorithms and consists of 100 game-rounds played by a human. Similar the *TCV100* data-set contains the recordings from the evaluation of our TCV algorithm. "*Both*" simply combines the last two, to create a more diverse data-set. For "*All*" we simply used all recordings we had available. Each data-set is randomly split into 90% training data and 10% development data.

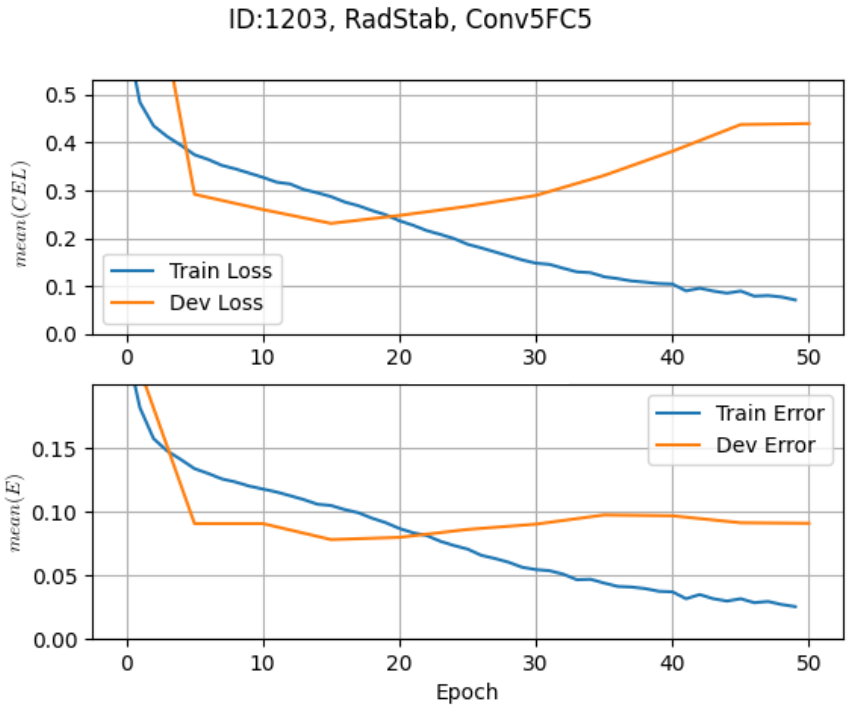


Figure 9: Exampe of the training loss and error for one trained configuration.

Name	Info	Nr. Training Samples	Nr Dev Dev samples
Human A	About 1.5h Human Play	22752	2527
Human100	100 rounds Human	18481	2053
TCV100	100 rounds PC	20418	2268
Both	Human100 $\cup$ TCV100	38898	4322
All	All above + $\approx$ 500TCV Games	161232	17914

Table 2: Data sets used for supervised training.

We also tested different neural network architectures, shown in Table 3. The simplest architecture is logistic regression. By adding additional layers, with 128 neurons per layer, we created fully connected networks with 1, 3, and 5 hidden layers. We then create convolutional networks always containing the same number of convolutional and fully-connected layers. The parameter count is decreased for *Conf2FC2* and *Conf3FC3*, because the reduction of the resolution in the first fully connected layer outweighs the newly added layers. *Confv2FC3* uses more filter kernels and higher neuron counts.

Name	Number of parameter
Logistic	69k
FC1	2 949k
FC3	2 982k
FC5	3 015k
Conv1Fc1	168k
Conv2FC2	153k
Conv3FC3	38k
Conv5FC5	709k
Conv2FC3	5 589k

Table 3: Parameter counts for different neural network architectures.

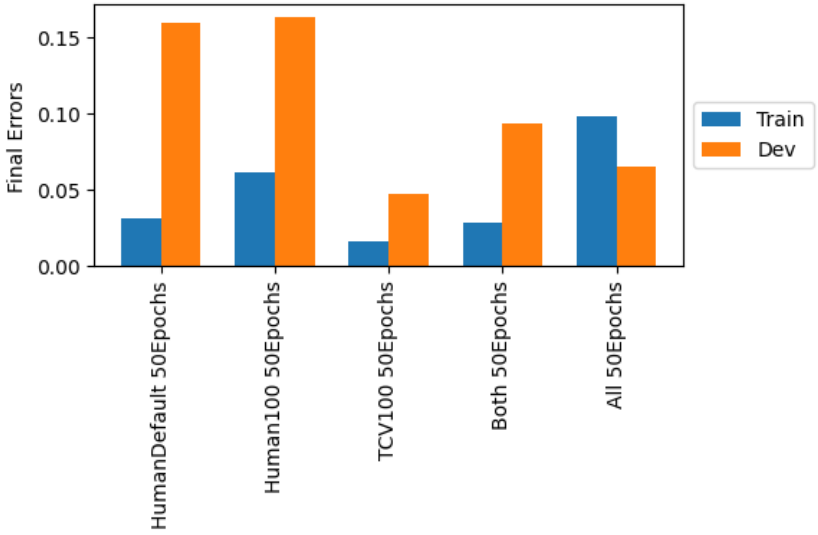
6.2 Results

In this section we evaluate the results based on the final training(train) and development(dev) errors as well as the in-game performance. We use the classification error $mean(E)$ as the ratio of wrong classified frames to all frames. The in-game performance is tested by playing 100 rounds and measuring the play-times.

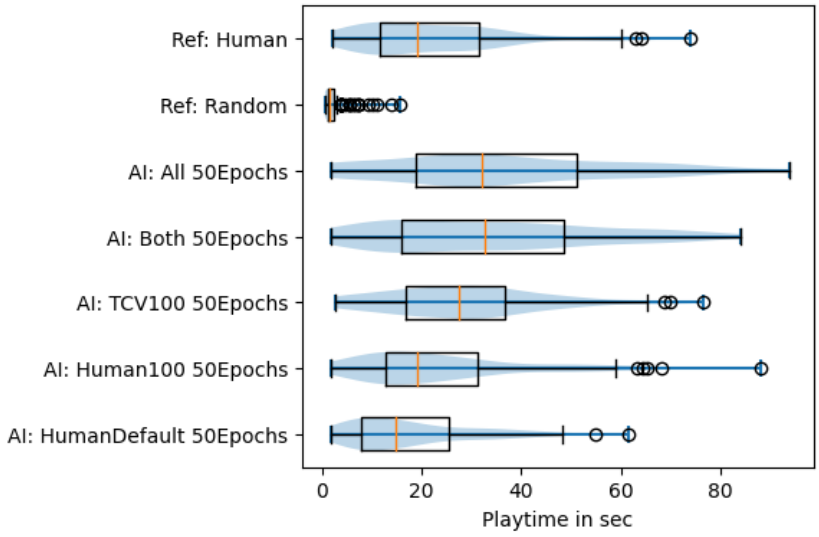
In Figure 10 we show the comparison of the different data-sets. We trained each of of them for 50 epochs. But final additional analyzes have shown a training of 10 to 20 epochs would be sufficient. The *Human* data-sets are harder to train. Most likely this is a result of the noise introduced by the reaction time. As the actions of the TCV algorithm are completely deterministic, they could be reproduced well by the CNN. The *both* data-set achieved the best in-game performance. The combination of human- and machine-data seams to have a positive effect. The hyperparameter choice was not optimal for the *All* data-set, which resulted in a high training error. The dev-set errors roughly match the results from the in-game evaluation. Based on the evaluation data we chose the *both* data-set for further evaluations.

Figure 11 compares different types of preprocessing and network architectures. In general, using fully connected networks resulted in a poor performance. The constant identical error, achieved by many different architectures, indicates a matching to some statistic property of the data-set. This also is supported by the results in Figure 12, where the fully connected networks performed comparable to random play. The chosen network architectures with 128 neurons per layer are most likely not ideal. The *radStab* preprocessing shows the best performance and generalizes better from the training- to the development-data. The *Conf5FC5* network also performed reasonable well on the configurations with less preprocessing.

A somewhat surprising result is the fact, that the best supervised learning results outperform the data-sets used for their training. The main factor for this is, that reaction time of the algorithm is faster than a human player. Also the combination of the human and TCV data-sets, which allows the algorithms to learn from both, contributes. On the other hand an inspection of the played games shows, that the algorithm has picked up some of the systematic errors from the TCV algorithm.



(a) Training results, $mean(E)$



(b) Evaluation results

Figure 10: Comparison of different data-sets.

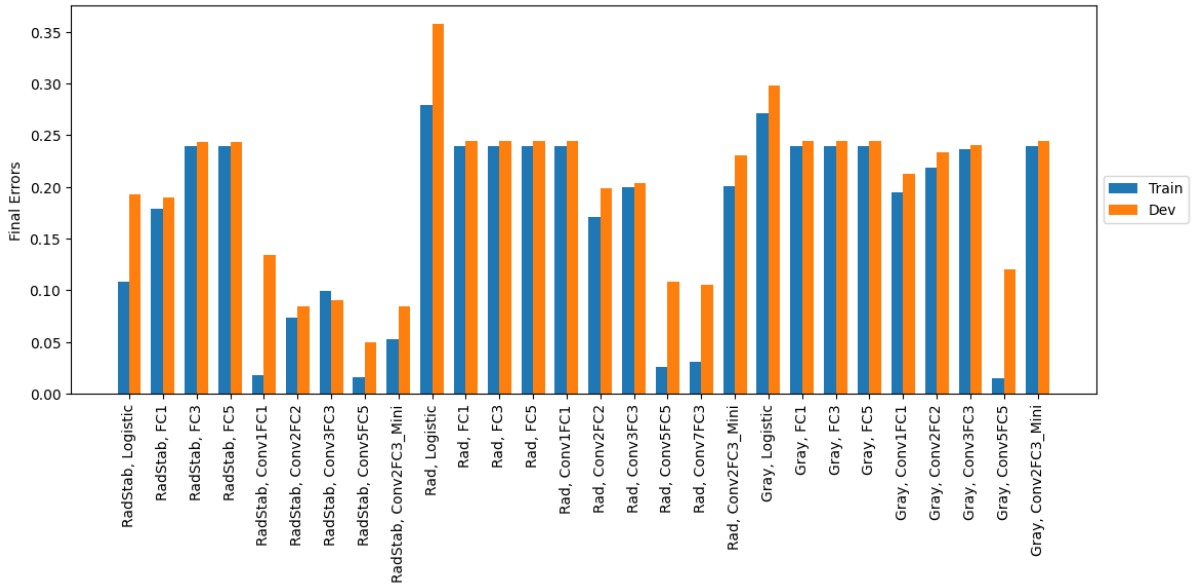


Figure 11: Comparisson of the classification errors ($mean(E)$) at the end of training.

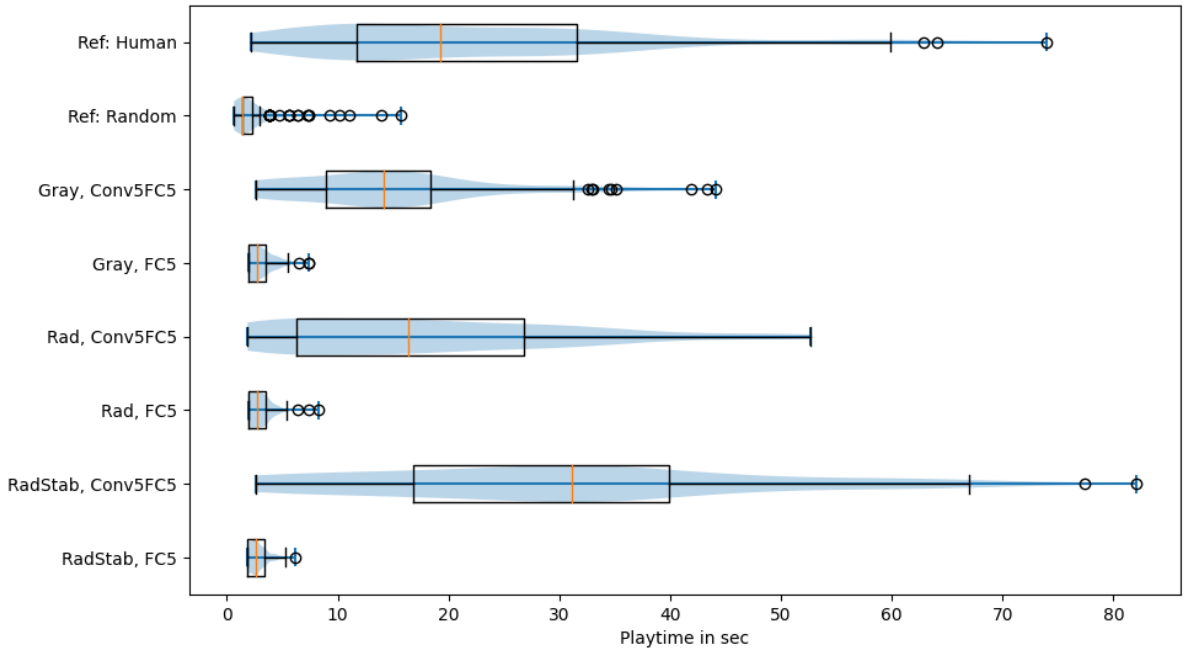


Figure 12: Playtimes of 100 rounds for different configurations.

7 Reinforcement Learning

The previously discussed implementations all have some intrinsic limitations. The TCV implementation needs to be updated for each situation that can happen in the game. The supervised implementation is limited by the training data and can outperform the training data only in terms of reaction time. Therefore the final goal is to implement a self-improving system based on reinforcement learning.

The foundation of reinforcement learning is a system consisting of an environment and an actor. The environment gets an action to perform from the actor. It executes the action and reports back a new state and a reward. The state is used by the agent to choose the next action. The reward is a feedback to the agent, how well it did, and is used to update its policy. The policy defines how the actor selects an action based on the current state. The different reinforcement algorithms now define how to change the policy to increase the expected reward. Generally we use the total discounted future reward according to Equation 16.

$$R_n = \sum_{i=0}^{N} \gamma^i * r_{n+i} \tag{16}$$

7.1 Algorithms

Textbooks usually start with table based methods. Most common is Q-Learning. Table-based Q-Learning stores a value for each possible pair of state and action. Sending raw images into the algorithm would need $nrValue^{nrPixels}$ rows and $nrActions$ columns. For us this would be $256^{192*120} * 3$ values. (This number has 55486 digits). Additionally each state must be visited multiple times. As a result table-based algorithms are only feasible in situations where the state space is small. Sometimes the input dimensions can to be reduced in a preprocessing step. In our case this could be done by threasholding and resizing. Using a binary image with 10x10 pixels results in 2^{10*10} states which is still not feasible. The other option to reduce the input space would be to extract some features and train on them. This again brings us in the direction of initial TCV implementation.

A common way to overcome this limitation is by approximating the Q-Table with a neural network. In the case of images usually a CNN. This type of network provides the capability of generalization. It can give us reasonable outputs for states, which it has never seen during training. This is the approach we use for this work. There are two common algorithms based on Q-Values, Q-Learing and SARSA, they are very similar in implementation. The difference is that Q-Learning always assumes using the best next step, while SARSA estimates, based on the actual step taken. This results in SARSA being less likely to take risks, which we consider favorable in our case. Additionally SARSA is usually more stable but requires a ε-decay to converge to the optimal policy.

The main alternative to the value methods like SARSA is policy gradient methods. They omit the prediction of the value and directly learn which action to perform in which state. A common implementation of this is the REINFORCE [12] algorithm. A combination of both groups are Actor Critic algorithms. They are basically policy gradient methods but use a value estimation to improve their learning.

7.2 SARSA with CNN

For this work we decided on using (Deep-)SARSA. It usually offers good performance and stability, while still being quite simple to implement. Further, being a value method, it is straightforward to debug. We also tested Deep Q-Learning but we had issues with convergence. We assume switching to a Dueling Q-Network [15] and playing with the learning rate would help.

The Deep in Deep-SARSA meaning we use a deep neural network. In our case we have a CNNs with 9 layers. The CNN architecture is guided by the results of our supervised tests. The base network architecture can be seen in Table 4 (Section 7.4). We also migrated from Tensorflow to PyTorch. Tensorflows Keras interface is a very nice abstraction for simpler supervised learning setups, but feels less helpful for reinforcement learning. PyTorch is usually just a bit more explicit about things.

SARSA is a acronym for the data needed for training (s_n,a_n,r_n,s_{n+1},a_{n+1}). A training sample consists of, state s and action a for the current timestamp (n) and next timestamp ($n+1$). Additionally the reward r which is receaved as a result of choosing action a_n is needed.

SARSA tries to approximate the Q-function ($Q^*(s_n, a_n)$) which is defined as expected future reward when the next action is a_n and then following actions are based on the current policy (Equation 17). In Equation 19 we now replace the unknown expected value by the outcome of the current sample and get a sample Q_s from the distribution. In Equation 20 we than estimate the expected future reward using our current Q function. Finally we update our $Q(s_n, a_n)$ function to minimize the overall error (Equation 21).

$$Q^*(s_n, a_n) \doteq \mathbb{E}(r_n|a_n) + \mathbb{E}(R(s_{n+1})) \tag{17}$$

$$Q^*(s_n, a_n) = \mathbb{E}(r_n|a_n) + \mathbb{E}(\sum_{i=0}^{\infty} \gamma^i r_{n+1+i}) \tag{18}$$

$$Q_s(s_n, a_n) = r_{n+1} + \gamma\mathbb{E}(\sum_{i=0}^{\infty} \gamma^i r_{n+1+i}) \tag{19}$$

$$Q_s(s_n, a_n) = r_{n+1} + \gamma Q_(S_{n+1}, A_{n+1}) \tag{20}$$

$$\sum |Q_s(s_n, a_n) - Q(s_n, a_n)| \rightarrow min \tag{21}$$

In most cases we use a ε-greedy policy. This delivers the best results. We also tested a greedy policy, which shows similar results. The big disadvantage of the greedy policy is, that it has a tendency to get stuck. For example it always makes the same move and crashes immediately. Other actions are never considered and as a result their values are not updated.

We need to choose an action for at least 3 to 5 times (depending on framerate) to see a meaningful effect. To address this we defined a special ε-greedy policy in Programm 1. It evaluates every 0.1sec if it should override the greedy policy. With a probability of ε it will start overriding, with a probability of 0.3 it will stop overriding. Also with a probability of 0.3 it will update its chosen action.

Program 1 Pseudo code of epsilon greedy policy

```
// Update policy override
each 0.1 sec
    if force=False:
        with probability ε:
            force := True
            force_action := random action
    if force=True:
        with probability 0.3:
            force := False
        else:
            with probability 0.3:
                force_action := random action
// execute policy
if force=true:
    a_n := force_action
else:
    a_n := π(s_n)
```

7.3 Architecture

7.3.1 Player and Trainer

We implemented the algorithm based on two separated programs, the Player and the Trainer. The architecture is illustrated in Figure 13. The Player plays the game, for each finished round it saves the recording to the file system. The Trainer reads a set of recordings and updates the policy based on it. The updated policy is then written to the file system. Because the only communication of player and trainer is the file system, they can be lunched on different PCs using a shared network file system. The game does not deliver any rewards, therefore the Trainer calculates the reward when the recording is incorporated into the replay buffer (Details in Section 7.3.3). Program 2 and 3 explain the main code path.

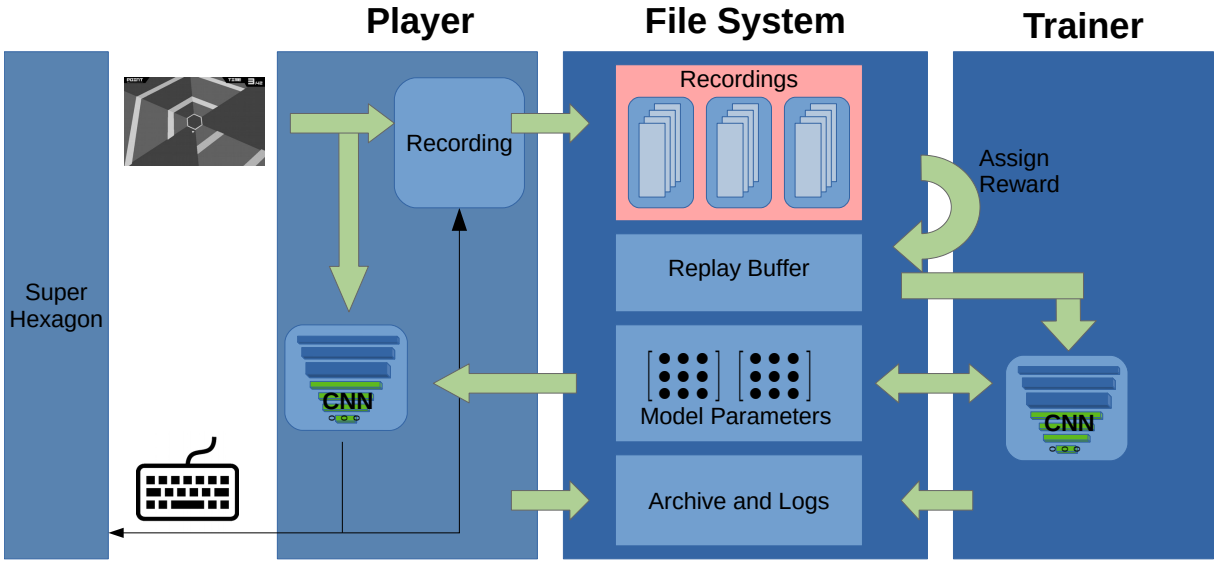


Figure 13: Software architecture used for training.

Program 2 Pseudo code of the Player

```
Start Game
forever:
    load newest policy
    play one round (using greedy polecy) // evaluation
    save Recording
    log result
    repeat M times:
        play one round (using e-greedy polecy) //training
        save Recording
        log result
```

Program 3 Pseudo code of the Trainer

```
    load model
    load replay buffer
    forever:
        // one epoch
        repeat N times:
            read Recording
            preprocess recording
            calculate reward
            add to replay buffer
        train CNN on all new samples
        train CNN on random old samples
        save model for player
```

7.3.2 Preprocessing

The preprocessing is based on the pipeline stages from the non-learning implementation. The preprocessors used are Gray Figure 4b ,Rad Figure 4c ,RadStab Figure 4f.

7.3.3 Reward

The reward functions consist of two parts. A positive and an negative reward. The negative reward is given for the last five frames before a collision. It is linearly faded on these frames (-20, -40, -60, -80, -100). The temporal extension of the reward to five frames should give more examples of what to avoid.

The positive reward is only given for some tested configurations. For these a distance is calculated on an ellipsoid according to Figure 14. This distance is then scaled to a range of 0 to *maxReward* (typically 3). The tests with just a negative reward are usually labeled *Neg*. The tests with negative and positive reward are labeled *Full*.

$$m[x,y] = 255\, max\left(0, 1 - \sqrt{\frac{x^2}{a^2} + \frac{(y - \frac{width}{2})^2}{b^2}}\right) | a = 1, b = 0.2, width = 400 \tag{22}$$

$$D = \max_{\forall x,y}(min(img(x,y), m(x,y))) \tag{23}$$

$$r = (255 - D)\frac{maxReward}{255} \tag{24}$$

Figure 14: The distance based reward is calculated using a distance template (red) which is combined with the thesholed image. The template is defined as an elypsoid.

7.3.4 Replay Buffer

The replay buffer is implemented as a hdf5 [6] file. The elements in the buffer are (State, Action, Reword) and RefFrame. RefFrame is a reference to the frame $0.1sec$ earlier. For training we allways use a temporal difference of $0.1sec$ between the two trained states. This is to overcome the systems lag, consecutive frames show basically no influence of the action of the direct predecessor. The actions values for this $0.1sec$ interval are averaged.

The replay buffer consists of three sub buffers D(Dummy), P(Positive), N(Negative). The D-buffer contains the first $1.2sec$ of the game-round. In this part of the game, the playfield is still mostly empty. The D-buffer only exists to be referenced by other elements. The N-Buffer contains the last second of each round. This is the interval just before the collision. The P-buffer contains the mid game, where no collisions happened. Generally we used a limit of 50k samples for P-Buffer and 150k buffer for N-buffer. These limits keep some balance of positive and negative samples with focus on the things to improve. Figure 15 shows the size development of the replay buffer during training. The P-buffer grows with an increasing speed as the algorithm improves. After about 300 epochs the P-buffer reaches its size limit and only the last 50k frames are used for training. The number of new samples per epoch also increases with performance and typically saddles at around 1k-4k samples per epoch.

Technically the replay buffer has no size limit, but only the last N samples are visible during training. The size limits of the replay buffer could be replaced by weighed sampling of the replay buffer.

7.4 The Model

The base model (Model1) used for our implementation has 5 convolution layers, 3 fully-connected hidden-layers and the output-layer. It converts a 192x120x1 tensor into a 3 element output. The main parameters are shown in Table 4. There are three variations of this model which have the last 10 actions as a secondary input. In Model2 the last 10 actions are added to the input of layer *fc2* and the last two actions are stacked as additional channels on top of the input of *Conf1*. In Model3 the last commands are added to the inputs of layers *fc1*, *fc2*, *fc3*. Model4 adds the actions to the inputs of layer *fc1*, *fc2*. Additionally Model 4 splits layer *fc3* and *fc4* in a value and advantage path which are added in the last step.

[6] https://www.hdfgroup.org/solutions/hdf5

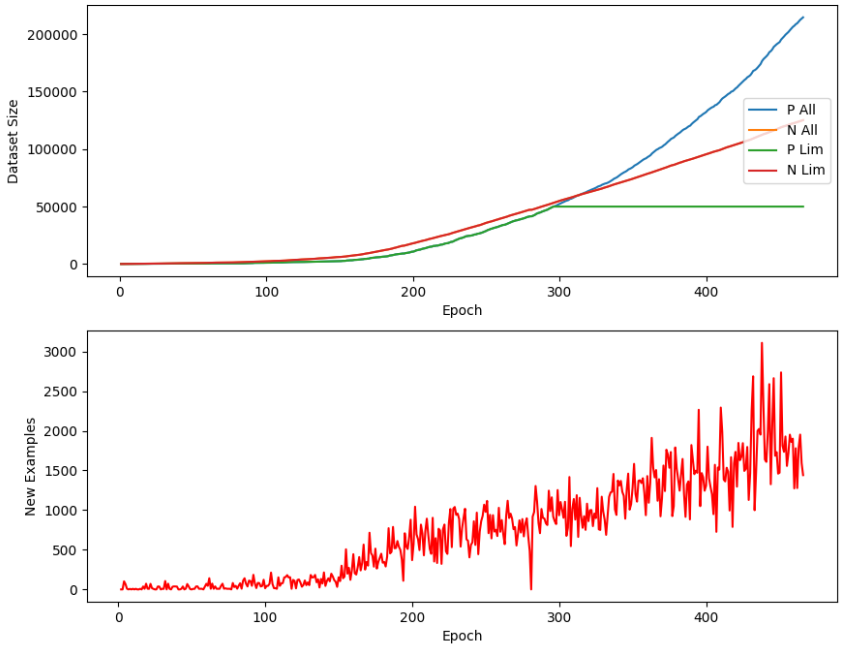


Figure 15: Size development of the replay buffer. The logical sizes are limited to 50k samples for P-buffer and 150k samples for N-buffer.

Layer	Type	Channels	kernel size/neurons	padding
Conf1	Conv2D	5	15	7
Conf11	Conv2D	9	9	2
Conf2	Conv2D	12	7	1
Conf3	Conv2D	15	7	1
Conf4	Conv2D	18	7	1
fc1	Fully Connected	-	128	-
fc2	Fully Connected	-	64	-
fc3	Fully Connected	-	24	-
fc4	Fully Connected	-	3	-

Table 4: Default CNN architecture used for our SARSA implementation (Model1). The activation function is exponential linear unit(elu) and all convolutional layers are followed by a MaxPooling.

7.5 Evaluation and Results

For the evaluation different configurations were trained. We trained each configuration for at least 10 hours. The most interesting configurations were trained further, up to 36 hours. An overview of the most relevant configurations and their key parameters can be seen in Table 5. Figure 16 shows results in training and evaluation mode. In evaluation-mode $\varepsilon = 0$, in training-mode ε is according to its configuration. The results can be split into two groups. All the Polar-Stab configurations, having the *radStab* preprocessing, performed well. The other preprocessors in comparison learned very slow. This is highlighted in Figure 17. The greedy configurations performed better during training and show no significant difference during evaluation. We also noticed, that smaller than default learning rates are most likely better. High learning rates as in configuration 9 resulted in a diverging optimization. Configuration 2 shows that using only negative feedback works too, but has slower improvements. The additional input of the last commands to the first layer of the CNN (Configuration 11) decreased performance. It was early stopped in favor of other configurations like 12p,13 which add the last commands only to the fully connected layers. We also experimented with resetting the state of the Adam optimizer between epochs (KM=0) but could not detect a significant influence. The best results were achieved by configuration 13 which was also used for comparison with other algorithms.

id	name	Model	Preprocessing	Reward	Additional Info
1	Polar-Stab Full	Model1	RadStab	Full	
2	Polar-Stab Neg	Model1	RadStab	Neg	$\alpha = 3e-5$ $beta = [0.99, 0.999]$ $KM = 1$
3	Polar Full	Model1	Polar	Full	
4	Gray Full	Model1	Gray	Full	
5	Gray Neg	Model1	Gray	Neg	
6	Polar-Stab Full Greedy	Model1	RadStab	Full	$\varepsilon = 0$
8	Polar-Stab Full Momentum	Model1	RadStab	Full	$KM = 1$
9	Polar-Stab Full Fast	Model2	RadStab	Full	$\alpha = 0.001$ $KM = 1$
10	Polar-Stab Full Slow	Model1	RadStab	Full	$\alpha = 0.00003$ $beta = [0.99, 0.999]$ $KM = 1$
11	Polar-Stab Full Hist	Model2	RadStab	Full	$\alpha = 0.00003$ $beta = [0.99, 0.999]$ $KM = 1$
12p	Polar-Stab Full Greedy2	Model3	RadStab	Full	$\alpha = 0.00003$ $beta = [0.99, 0.999]$ $KM = 1$
13	Polar-Stab Full optim	Model4	RadStab	Full	$\alpha = 0.00003$ $beta = [0.99, 0.999]$ $sizeP = sizeN = 200k$ $KM = 1$ $\varepsilon = decay$

Table 5: Configurations used for evaluation. Defaults are learning rate $\alpha = 1e-4$, momentum $beta = [0.9, 0.999]$, size of P-buffer $sizeP = 50k$, size of N-buffer $sizeN = 150k$, keep momentum $KM = 0$, randomness during training $\varepsilon = 0.03$

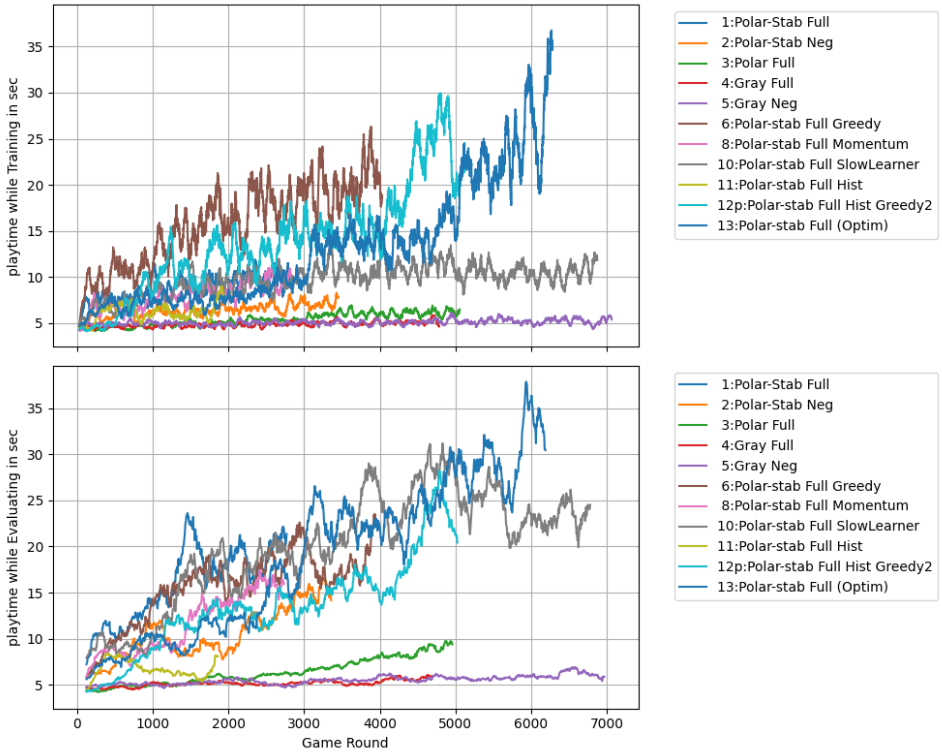


Figure 16: Playtimes in evaluation mode($\varepsilon = 0$) and training mode($\varepsilon \neq 0$). Rolling average over 50 rounds.

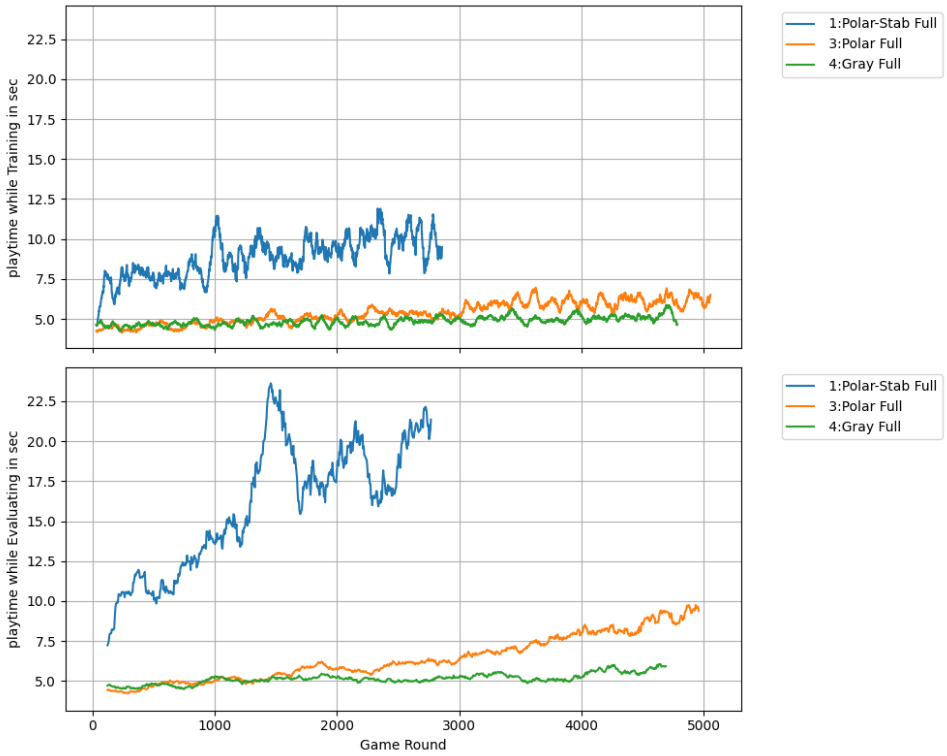


Figure 17: Selection of curves from Figure 16. Highlighting the influence of preprocessing.

8 Conclusion

We have implemented an artificial player for Super Hexagon using three different methods. We compared each one against a human reference and analyzed the impact of preprocessing and other parameters. Now we compare the different algorithms, Table 6 shows the values and Figure 18 visualizes their distributions.

With 24.4sec the Traditional Computer Vision (TCV) implementation was able to slightly surpass human performance. The implementation lead to the development of our preprocessing pipeline from which all other algorithms could benefit. For sure there is still a significant room for improvement, but driving this approach forward is very labor intensive.

The supervised learning implementation is able to reproduce and even surpass its training data. This mostly is attributed to the superhuman reaction-times with an additional small boost from combining the Human- and TCV-data for training. Based on the sophisticated frameworks this approach was straightforward to implement, this mostly moved the development efforts towards the evaluation.

The reinforcement learning implementation needs significantly more application specific tweaking, especially when it comes to systems running at real time. Nevertheless the existing frameworks support the implementation very well, allowing the developer mostly to focus on application specific tasks. The reinforcement system significantly outperformed all other implementations and most likely improves even further when training is continued.

The main benefit of TCV in our perspective is the possibility to reason about its actions. If this is not necessary, we suggest a learning based approach. A supervised system will quickly lead to good results if training data in sufficient amount and quality is available. If the best possible performance is your goal, a reinforcement system is the way to go.

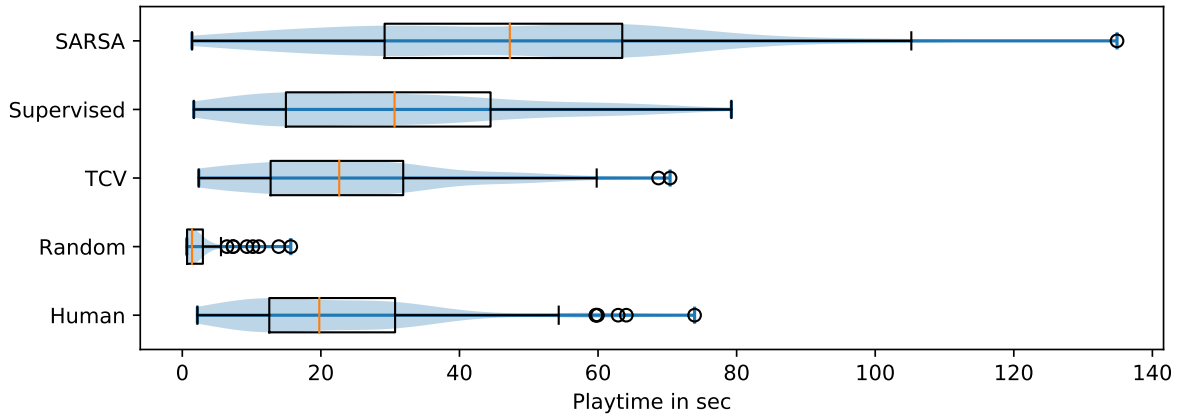


Figure 18: Comparison of 100 game-rounds of different approaches.

	Mean	Std-Dev	Median	Max
SARSA	46.9	25.6	47.3	134.9
Supervised	30.7	19.2	30.6	79.2
TCV	24.4	15.4	23.5	70.4
Random	2.5	2.7	1.4	15.7
Human	22.9	15.2	19.7	73.9

Table 6: Result comparison of different algorithms. All values are play-times in seconds.

References

[1] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000. URL: https://opencv.org/. 5

[2] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016. arXiv:arXiv:1606.01540. 1, 2

[3] Danijar Hafner. Deep reinforcement learning from raw pixels in doom. *CoRR*, abs/1610.02164, 2016. URL: http://arxiv.org/abs/1610.02164. 1

[4] Todd Hester, Matej Vecerík, Olivier Pietquin, Marc Lanctot, Tom Schaul, Bilal Piot, Andrew Sendonaris, Gabriel Dulac-Arnold, Ian Osband, John P. Agapiou, Joel Z. Leibo, and Audrunas Gruslys. Learning from demonstrations for real world reinforcement learning. *CoRR*, abs/1704.03732, 2017. URL: http://arxiv.org/abs/1704.03732. 1

[5] Michal Kempka, Marek Wydmuch, Grzegorz Runc, Jakub Toczek, and Wojciech Jaskowski. Vizdoom: A doom-based AI research platform for visual reinforcement learning. *CoRR*, abs/1605.02097, 2016. URL: http://arxiv.org/abs/1605.02097. 1

[6] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017. arXiv:1412.6980. 11

[7] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. *CoRR*, abs/1602.01783, 2016. URL: http://arxiv.org/abs/1602.01783. 1

[8] Somnuk Phon-Amnuaisuk. Learning to play pong using policy gradient learning. *CoRR*, abs/1807.08452, 2018. URL: http://arxiv.org/abs/1807.08452, arXiv:1807.08452. 1

[9] Deep Reinforcement Learning: Pong from Pixels. URL: http://karpathy.github.io/2016/05/31/rl/. 1

[10] Building a Powerful DQN in TensorFlow 2.0 (explanation & tutorial). URL: https://medium.com/analytics-vidhya/building-a-powerful-dqn-in-tensorflow-2-0-explanation-tutorial-d48ea8f3177a. 1

[11] Simon Ramstedt and Christopher J. Pal. Real-time reinforcement learning. *CoRR*, abs/1911.04448, 2019. URL: http://arxiv.org/abs/1911.04448, arXiv:1911.04448. 1

[12] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018. URL: http://incompleteideas.net/book/the-book-2nd.html. 1, 16

[13] Jaden B. Travnik, Kory W. Mathewson, Richard S. Sutton, and Patrick M. Pilarski. Reactive reinforcement learning in asynchronous environments. *Frontiers in Robotics and AI*, 5:79, 2018. URL: https://www.frontiersin.org/article/10.3389/frobt.2018.00079, doi:10.3389/frobt.2018.00079. 1

[14] Ziyu Wang, Nando de Freitas, and Marc Lanctot. Dueling network architectures for deep reinforcement learning. *CoRR*, abs/1511.06581, 2015. URL: http://arxiv.org/abs/1511.06581. 1

[15] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, and Nando de Freitas. Dueling network architectures for deep reinforcement learning, 2016. arXiv:1511.06581. 17

Acronyms

HSV Hue Saturation Value

CNN Convolutional Neural Network

SARSA State Action Reward State Action

TCV Traditional Computer Vision

SAC Soft Actor Critic

NEAT Neuro Evolution of Augmenting Topologies

API Application Programming Interface

fps frames per second

CEL Cross Entropy Loss